

Permissive Supervisor Synthesis for Markov Decision Processes through Learning

Bo Wu, Xiaobin Zhang and Hai Lin

Abstract—This paper considers the permissive supervisor synthesis for probabilistic systems modeled as Markov Decision Processes (MDP). Such systems are prevalent in power grids, transportation networks, communication networks and robotics. Unlike centralized planning and optimization based planning, we propose a novel supervisor synthesis framework based on learning and compositional model checking to generate permissive local supervisors in a distributed manner. With the recent advance in assume-guarantee reasoning verification for probabilistic systems, building the composed system can be avoided to alleviate the state space explosion and our framework learn the supervisors iteratively based on the counterexamples from verification. Our approach is guaranteed to terminate in finite steps and to be correct.

Index Terms—formal methods, supervisor synthesis, model checking, automata learning

I. INTRODUCTION

Control and planning of probabilistic systems modeled as Markov Decision Processes (MDP) [1] has gained tremendous popularity in recent years because it not only considers the uncertainties in actuation and environments for each individual agent but also the coordination among them for achieving a global specification. The development of a group of several specialized agents could offer better efficiency, fault-tolerance, and flexibility due to its distributed operations, parallelized executions and reconfigurability in face of changes.

Since multiple MDPs can be composed into one, a centralized planning approach is the probabilistic model checking [2] to obtain the global optimal strategy (policy) for single MDP such as in [3], [4] with specifications in probabilistic linear temporal logic (PLTL) [5] and computation tree logic (PCTL) [6]. However, a global controller supervising all the agents may not be feasible. Another planning approach is based on the Decentralized MDP (Dec-MDPs) [7] where each agent make decisions with its local information. Most existing results focus on seeking the (approximately) optimal action policies through reinforcement learning or Q-learning [8], or alternatively converting the problem into stochastic games (SG) [9] but such solutions can be computationally expensive and often intractable if not undecidable.

This paper aims to develop a formal synthesis framework with MDPs so that a certain desired performance can be guaranteed with the designed local supervisors. Without loss of optimality in the bounded time properties that we consider [10], the supervisors are in the form of the deterministic finite automata (DFA) [11] that regulate each agent's action.

Unlike the optimization based approach which searches for a single optimized control policy, we are looking for *permissive* local supervisors that potentially enable multiple choices in each step so that the system is resilient against unexpected changes in the runtime or additional constraints may be imposed on the system. Similar ideas was studied in [12] where memoryless (making decision based only on current state) permissive strategies were generated incrementally using mixed integer linear programming (MILP) [13] for the single agent. The agent and its environment were modeled as turn based stochastic two-player games. In contrast, our approach considers strategies with memory and works in a decremental fashion by iteratively eliminating counterexample strategies from model checking and refining the local supervisor with L* learning algorithm [14].

In recent years, counterexample guided approaches have been extensively studied in the aspect of abstraction refinement [15], [16] and synthesis [17], [18], [19]. In this paper, we mainly focus on the latter. The main purpose of the counterexample guided synthesis is to iteratively learn a correct supervisor under whose regulation the system is guaranteed to satisfy a given specification. In each intermediate iteration, a guessed supervisor is automatically generated whose correctness will be verified by the model checker and other oracles and refined based on the counterexamples from verification. Such verification-refinement loop ends when no counterexample is generated and thus the guessed supervisor is verified to be correct.

Synthesis of multiple agents is not straightforward for systems modeled as MDPs. Unlike the centralized probabilistic model checking approach, we propose to apply the existing assume guarantee reasoning (AGR) framework for the probabilistic systems [15], [20], [21] to alleviate the computation burden. While our framework does not require a particular assume-guarantee reasoning algorithm, we can choose any sound and complete assume-guarantee reasoning algorithms and potentially benefit from state space reduction during the model checking process. The second challenge is to process the counterexamples returned from the compositional model checking. To this end, we propose to label the actions such that we can identify which individual agent causes the specification violation. L* learning is adopted to each single agent based on the counterexample it receives to iteratively refine the corresponding local supervisor. The correctness and termination of our design procedure are proved.

This paper merges and further develops our previous results [22], [23]. While the same L* learning algorithm [14] was considered in synthesis process, we changed the form of the supervisor and considered positive counterexamples so that the results can become less conservative. We also changed

This work is supported by NSF-CNS-1239222, NSF-EECS-1253488 and NSF-CNS-1446288

Bo Wu, Xiaobin Zhang and Hai Lin are with the Department of Electrical Engineering, University of Notre Dame, Notre Dame, IN, 46556 USA. bwu3@nd.edu, xzhang11@nd.edu, hlin1@nd.edu

the counterexample selection scheme to potentially lower the complexity. Furthermore, we developed the supervisor synthesis software in C++ using counterexample generation tool COMICS [24] and L* learning library libalf [25].

This paper is divided into five parts. We give the necessary preliminaries in Section II followed by problem formulation in Section III. We describe our proposed supervisor synthesis framework in Section IV. Section V concludes the paper.

II. BACKGROUND

This section introduces the required background knowledge about MDP, model checking [2], and the L* algorithm.

A. Markov Decision Process

Definition An MDP is a tuple $\mathcal{M} = (S, \hat{s}, A, T, L)$ where

- $S = \{s_0, s_1, \dots\}$ is a finite set of states;
- $\hat{s} \in S$ is the initial state;
- A is a finite set of actions;
- $T(s, a, s') := \Pr(s'|s, a), \forall i \geq 0$.
- $L : S \rightarrow 2^{AP}$ is the labeling function that maps each $s \in S$ to one or several elements of a set AP of atomic propositions.

For each state $s \in S$, we denote $A(s)$ as the set of available actions. From the definition it is not hard to see that the Discrete Time Markov Chain (DTMC) is a special case of MDP with $|A(s)| = 1$ for all $s \in S$, where $|A(s)|$ is the cardinality of the set $A(s)$.

A path ω of an MDP is a non-empty sequence of the form $\omega = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \dots s_i \xrightarrow{a_i} s_{i+1} \dots$, where each transition is enabled by an action a_i such that $T(s_i, a_i, s_{i+1}) > 0$. We denote $Path_s^{fin}$ as the collection of finite length paths that start in a state s . We also denote ω^a as the action path that's embedded in ω . The nondeterminism of an MDP is resolved with the help of a scheduler.

Definition A scheduler σ (also known as adversary or policy) of an MDP \mathcal{M} is a function mapping every finite path ω_{fin} onto an action $a \in A(last(\omega_{fin}))$ where $last(\omega_{fin})$ denotes the last state of ω_{fin} .

The scheduler σ specifies the next action for each finite path. The behavior of an MDP \mathcal{M} under a given scheduler σ is purely probabilistic and thus reduces to a DTMC \mathcal{M}_σ . We denote $\Sigma_{\mathcal{M}}$ as the (possibly infinite) set of all possible schedulers for \mathcal{M} . There is a one-to-one correspondence between the paths of the DTMC and that of the MDP.

B. Probabilistic Model Checking

With the MDP model defined above, we can then determine if a given MDP satisfies some specification ϕ . For the specification we use the Probabilistic Computation Tree Logic (PCTL) [1] that can be seen as a probabilistic extension of Computation Tree Logic (CTL). The PCTL formulas we consider in this paper are the time-bounded weak-safety fragment [26].

Definition The syntax of a weak-safety PCTL is defined as $\phi ::= true \mid a \mid \phi \vee \wedge \phi \mid \mathcal{P}_{\leq p}[X\psi] \mid \mathcal{P}_{\leq p}[\psi \mathcal{U}^{\leq k} \psi]$;

$\psi ::= true \mid a \mid \psi \vee \wedge \psi \mid \neg(\mathcal{P}_{\leq p}[X\psi]) \mid \neg(\mathcal{P}_{\leq p}[\psi \mathcal{U}^{\leq k} \psi])$, where ϕ is the weak safety fragment and ψ is the strict liveness fragment, $a \in AP$, $p \in [0, 1]$, X for "next", $\mathcal{U}^{\leq k}$ for "bounded until" with $k \in \mathbb{N}$.

Satisfaction of a PCTL formula $\mathcal{P}_{\leq p}(\varphi)$ means that the probability of satisfying φ fulfills the comparison $\leq p$ over all possible schedulers. The purpose of the probabilistic model checking is to give a Boolean answer to $\mathcal{M} \models \mathcal{P}_{\leq p}(\varphi)$.

C. L* learning algorithm

The L* learning algorithm [14] is one of widely used online learning algorithms for regular languages. It basically learns a *minimal* Deterministic Finite Automata (DFA) that accepts an unknown regular language \mathcal{L} by interacting with a *teacher*. The definitions of DFA and language are described below.

Definition A *finite automaton* (FA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet of actions, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of *accepting states*. \mathcal{A} is called *deterministic* (DFA) if $|Q_0| \leq 1$ and $|\delta(q, a)| \leq 1$ for all states $q \in Q$ and $a \in \Sigma$.

Definition A word $s = a_1 a_2 \dots a_n$ where $a_i \in \Sigma$ is accepted by an FA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ if it induces a run $q_0 q_1, \dots, q_n$ on FA where $q_i \in Q$ and $q_n \in F$. The collection of all finite words $S \in \Sigma^*$ accepted by \mathcal{A} is called the language accepted by \mathcal{A} denoted as $\mathcal{L}(\mathcal{A})$.

The teacher answers two types of questions, namely the *membership* query and the *conjecture*. The membership query returns true if some words belong to the target language and false otherwise. For the conjecture, if the hypothesized DFA accepts the target language, it will return true and the learning process is finished. Otherwise, the teacher must return a counterexample to illustrate the difference between the conjectured DFA and the unknown plant.

During the learning process, the L* algorithm maintains a so-called *observation table* (S, E, T) where $S \subseteq \Sigma^*$ is a set of prefixes, $E \subseteq \Sigma^*$ is a set of suffixes and $T : (S \cup S \cdot \Sigma) \times E \rightarrow \{0, 1\}$. Σ^* denotes finite traces from the alphabet set Σ . For $s \in S \cup S \cdot \Sigma$, $e \in E$, if $s \cdot e \in \mathcal{L}$, then $T(s, e) = 1$ and $T(s, e) = 0$ if $s \cdot e \notin \mathcal{L}$. The L* algorithm will always keep the observation table *closed and consistent* as defined in [14]. It is guaranteed to learn a minimal DFA with $(|\Sigma|n^2 + n \log m)$ membership queries and at most $n - 1$ conjectures, where n is the number of states in the final DFA and m is the length of the longest counterexample in conjectures [14]. Due to space limit we will not elaborate the details here.

III. PROBLEM FORMULATION

Our target system model is an MAS consists of N agents modeled as MDPs $\mathcal{M}_i, i \in [1, N]$. Define \mathcal{M} as the composition of N subsystems such that $\mathcal{M} = \mathcal{M}_1 || \mathcal{M}_2 || \dots || \mathcal{M}_N$.

The parallel composition of MDPs are defined as below:

Definition Given two Markov decision processes $\mathcal{M}_1 = (S_1, s_0^1, A_1, T_1, L_1)$ and $\mathcal{M}_2 = (S_2, s_0^2, A_2, T_2, L_2)$, the parallel composition of \mathcal{M}_1 and \mathcal{M}_2 is an MDP $\mathcal{M} = \mathcal{M}_1 || \mathcal{M}_2 =$

$(S_1 \times S_2, s_0^1 \times s_0^2, A_1 \cup A_2, T, L)$ where $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$ and

- $T((s_1, s_2), a, (s'_1, s'_2)) = T_1(s_1, a, s'_1)T_2(s_2, a, s'_2)$ if $a \in A_1 \cap A_2$ and both $T_1(s_1, a, s'_1)$ and $T_2(s_2, a, s'_2)$ are defined, or
- $T((s_1, s_2), a, (s'_1, s'_2)) = T_1(s_1, a, s'_1)$ if $a \in A_1 \setminus A_2$ and $T_1(s_1, a, s'_1)$ is defined, or
- $T((s_1, s_2), a, (s'_1, s'_2)) = T_2(s_2, a, s'_2)$ if $a \in A_2 \setminus A_1$ and $T_2(s_2, a, s'_2)$ is defined.

The probabilistic specification is given as $\phi = \mathcal{P}_{\leq p}(\varphi)$ where $\varphi = \phi_1 U^{\leq k} \phi_2$. It is a time-bounded weak-safety PCTL to guarantee certain properties. The reason why we focus on timed bounded formulas is that for many practical systems, because of power constraint and practical limitation like memory, the observation duration is often of limited time. Our aim is to determine if $\mathcal{M} \models \phi$. If not, we need to synthesize local supervisors $\mathcal{K}_i, i \in [1, N]$ that dynamically disable certain actions $a_i \in A_i$ so that the controlled system \mathcal{M} can satisfy the specification ϕ .

In particular, the supervisors $\mathcal{K}_i, i \in [1, N]$ is the form of DFAs with the alphabet set being given as $\Sigma_i = S_i \times A_i$. we define another notion of parallel composition \parallel_{sup} to illustrate how the supervisors regulate \mathcal{M} .

Definition Given an MDP $\mathcal{M} = (S, s_0, A, T, L)$, and the DFA $\mathcal{K} = \{Q, \Sigma, \delta, q_0, Q_m\}$, the parallel composition between \mathcal{M} and \mathcal{K} is an MDP $\mathcal{P} = \mathcal{M} \parallel_{sup} \mathcal{K}$:

- $S^{\mathcal{P}} = \{(s, q) | s \in S, q \in Q\}$ is a finite set of states;
- (s_0, q_0) is the initial state;
- $A^{\mathcal{P}} = A$ is a finite set of actions;
- $T^{\mathcal{P}}((s, q), a, (s', q')) := T(s, a, s')$, if $\delta(q, sa) = q'$;

There is a one-to-one correspondence between a path from \mathcal{P} , $\omega' = (s_0, q_0) \xrightarrow{a_0} (s_1, q_1) \xrightarrow{a_1} (s_2, q_2) \dots (s_i, q_i) \xrightarrow{a_i} (s_{i+1}, q_{i+1})$ to the path in the original MDP \mathcal{M} , $\omega = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \dots s_i \xrightarrow{a_i} s_{i+1}$. From the definition of \parallel_{sup} , only those transitions that are defined on both \mathcal{M} and \mathcal{K} are enabled, therefore the supervisor controls what actions can be allowed on each state. We assume that any $a_i \in A_i, \forall i \in [1, N]$ is controllable meaning that we could disable or enable it if we want. Note that in our previous results [22], [23], the local supervisor's alphabet $\Sigma_i = A_i$. From the definition of the parallel composition \parallel , it can be seen that it only keeps track of the action sequence but not state sequence. So our previous supervisor eliminates all the path with the same action sequence while in this paper, the supervisor keeps track of the state as well and thus is less conservative.

Formally, our permissive supervisor synthesis problem can be formulated as follows:

Problem 1 Given an MAS consisting of N system models $\mathcal{M}_i = (S_i, s_{0,i}, A_i, T_i, L_i), i = 1, \dots, N$ and a probabilistic specification ϕ represented by PCTL such that $\mathcal{M} \not\models \phi$, where $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2 \parallel \dots \parallel \mathcal{M}_N$. The supervisor synthesis problem is to automatically synthesize N local supervisors \mathcal{K}_i such that $\tilde{\mathcal{M}} \models \phi$, where $\tilde{\mathcal{M}} = \tilde{\mathcal{M}}_1 \parallel \tilde{\mathcal{M}}_2 \parallel \dots \parallel \tilde{\mathcal{M}}_N$ and $\tilde{\mathcal{M}}_i = \mathcal{M}_i \parallel_{sup} \mathcal{K}_i, i = 1, \dots, N$.

IV. COUNTEREXAMPLE-GUIDED SUPERVISOR SYNTHESIS FRAMEWORK

In this section, we will introduce the proposed counterexample guided supervisor synthesis framework. But before that, we need to first introduce the counterexamples in an MDP model.

A. Counterexamples

Counterexamples are one of the most important features in model checking which illustrates how one model violates certain property. They are the key ingredients in counterexample guided synthesis [17] and abstraction refinement [27]. The form of counterexample varies by the checked formula and the given model. In non-probabilistic systems, for example safety properties like *bad things should never happen* can be simply refuted by a single path that leads to a bad state. However, in probabilistic models, counterexample generation is non-trivial and is an active research area. The main difficulty is that, while one single path may be valid by obeying the probability bound under given scheduler, a collection of the finite paths may violate the probabilistic specification simply because their accumulative probability exceeds the desired bound.

For MDP models, since only with given scheduler can the MDP have probability measure, our first step is to find out the scheduler under which the specification ϕ is violated. In this case the MDP will be reduced to a DTMC. Then several results exist to generate counterexample in DTMC, for example, k shortest path algorithm [28] or minimal critical subsystem generation from [29].

B. Single Agent Case

A summary of our proposed supervisor synthesis framework is illustrated in Algorithm 1. Given an MDP model \mathcal{M} and a probabilistic specification ϕ , there are two tasks to be done in each iteration where \mathcal{K}^i and $\tilde{\mathcal{M}}^i$ are the resulting supervisor and the composed system respectively after the i -th iteration.

i) Verification as shown from line 3 to line 7 of Algorithm 1. In the first model checking ($i = 0$), if $P_{min} \leq p$, we can extract the strategy σ_{min} as a DFA for later use. Given $\tilde{\mathcal{M}}^i$ and the probabilistic property $\phi = \mathcal{P}_{\leq p}(\varphi)$ where $\varphi = \phi_1 U^{\leq t} \phi_2$, this phase first checks whether $\tilde{\mathcal{M}}^i \models \phi$. If not, a DTMC T^i induced from a scheduler will be returned as the counterexample to select the counterexample path π^i . Else if there is positive counterexample, it will be saved to π^i . If no counterexample in both cases, then we are done.

ii) Learning based synthesis. A synthesis-specific implementation of the L^* learning algorithm is proposed here. In this stage, to be able to answer the membership query, we will view \mathcal{M} as a finite automaton (FA) by ignoring its probabilistic attributes and leaving only its states, transition relation and labeling. Formally, we will reduce $\mathcal{M} = (Q, q_0, A, Steps, L)$ into a labeled FA $\mathcal{M}_D = (Q, A, \delta, q_0, F, L)$ where for all $q, q' \in Q$ and $a \in A$, $\delta(q, a) = q'$ if and only if there is an action-distribution pair $(a, \mu) \in Steps(q)$ and $\mu(q') > 0$. The accepting states $F = Q$. Since we only consider a bounded time k , any path longer than k doesn't matter to the supervisor and we answer them with 1.

Algorithm 1: SPVSYN(\mathcal{M}, ϕ)

input : An MDP model \mathcal{M} , probabilistic specification $\phi = \mathcal{P}_{\leq p}(\varphi)$
output: \mathcal{K}^i such that $\widetilde{\mathcal{M}}^i = \mathcal{M}||_{sup}\mathcal{K}^i \models \phi$ or false if there does not exist such supervisor.

```

1  $i \leftarrow 0, \mathcal{K}^i = \text{DFA } \mathcal{G} \text{ such that } \mathcal{L}(\mathcal{G}) = \Sigma^*, \widetilde{\mathcal{M}}^i = \mathcal{M};$ 
2 while true do
3   if ModelChecking( $\widetilde{\mathcal{M}}^i, \phi$ )=false then
4     if  $i=0$  &  $P_{min}(\varphi) > p$  then
5       return false;
6     end
7      $\widetilde{\mathcal{M}}_{\sigma_i} \leftarrow$  the counterexample DTMC;
8      $\pi^i \leftarrow \text{HSP}(\widetilde{\mathcal{M}}_{\sigma_i}, \phi);$ 
9     else if Positive counterexample exists then
10     $\pi^i \leftarrow$  Positive counterexample;
11  else
12    Break;
13   $\mathcal{K}^{i+1} \leftarrow \text{L}^* (\pi^i), \widetilde{\mathcal{M}}^{i+1} = \mathcal{M}||_{sup}\mathcal{K}^{i+1};$ 
14   $i \leftarrow i + 1;$ 
15 end
16 return  $\mathcal{K}^i;$ 

```

We then describe the learning process in more detail. It makes use of the L^* learning as a template algorithm to learn the unknown supervisor \mathcal{K} . The teacher to membership query will be \mathcal{M}_D which tells if some finite state action path belongs to it or not. After some membership queries, \mathcal{K}^i will be generated as in line 7 in Algorithm 1. The conjecture query is answered by probabilistic model checking as shown in line 3 to line 6 in Algorithm 1 for negative counterexamples (those should be disallowed but currently allowed by the supervisor) and \mathcal{M}_D together with the σ_{min} for positive counterexamples (those should be allowed but currently absent in the supervisor). If the probabilistic model checking returns false, it will return DTMC T^i as in line 5 and a state action path π^i will be selected as the counterexample for the conjecture as in line 6. If positive counterexample exists, it will be passed to L^* learning as in line 7. In line 8, the L^* learning algorithm will update its observation table accordingly and run another round of membership query to conjecture the supervisor. To be permissive, for $i = 0$ every possible path is allowed. After that, \mathcal{K}^i where $i \geq 1$ will be conjectured iteratively until neither positive nor negative counterexample is generated. Then the supervisor is obtained by eliminating all its transitions to the non-accepting states.

A notable difference of our modified L^* learning algorithm from the traditional one is that in our framework, the answer to the membership query might be changed by the counterexamples returned from the conjecture, while in traditional L^* learning the answer to the membership query will remain the same throughout the learning process. The reason is that in our case, as we are looking for a permissive supervisor, when some finite path ω is accepted by the \mathcal{M}_D , our membership query will return true and our supervisor will allow it to happen first. But it could be the case that $\omega \in \pi^i$ at some

iteration i later which means that it is actually found to be a counterexample path and should be eliminated. Then the answer to this membership query will be changed from true to false and the observation table will be again checked for closedness and consistency and may bring several additional rounds of membership queries.

1) *Counterexample selection:* When $\widetilde{\mathcal{M}}^i \not\models \phi$ where $\phi = \mathcal{P}_{\leq p}(\varphi)$, it will return the DTMC $\widetilde{\mathcal{M}}_{\sigma_i}$ induced by the optimal scheduler σ_i on $\widetilde{\mathcal{M}}^i$ such that $P_{\widetilde{\mathcal{M}}_{\sigma_i}}(\varphi) > p$. Then this stage is responsible for selecting a particular counterexample path π^i to eliminate. We could apply the hop-constrained shortest path (HSP) algorithm[30] to find the path with the largest probability that is not induced by σ_{min} . Since σ_{min} is not a counterexample strategy, such path is guaranteed to exist. The resulting π^i will then be passed to the third stage to answer the conjecture in our modified L^* learning algorithm. Note that in our previous work [22], [23], we proposed to apply the hop-constrained k shortest path (HKSP) algorithm to find the smallest k paths so that eliminating them will make sure the DTMC will not be counterexample any more. The complexity of HKSP is higher than the HSP since HKSP has to search for k as well. Additionally, any state action path with length no larger than t that exists in the original MDP but not in the supervised MDP and none of the counterexample paths is its prefix can serve as a positive counterexample. Such path can be found by computing the difference between the regular languages generated by the underlying finite automata (with the state action pair as alphabet) of the MDP and the supervised MDP.

2) *Correctness and termination:*

Assumption 1. For each $s \in S$ from \mathcal{M} , the support for every distribution μ_s^a where $\mu_s^a(s') = T(s, a, s')$ are the same.

Each iteration of synthesis flow will terminate because \mathcal{K} is a DFA and L^* learning algorithm learns a minimal DFA in a finite number of queries [14]. Theorem 2 proves that the result of supervisor synthesis is correct by design and always terminates.

Theorem 1. If the supervisor exists, the synthesized supervisor \mathcal{K} parallel composes with original MDP \mathcal{M} satisfies the specification ϕ and the whole synthesis process also terminates in finite iterations and the supervisor is nonblocking.

Proof. If $\mathcal{M} \models \phi$, the synthesis terminates when $i = 0$ and the theorem trivially holds. Otherwise, for probabilistic property in the form of $\phi = \mathcal{P}_{\leq p}(\varphi)$ where $\varphi = \phi_1 U^{\leq t} \phi_2$, the number of possible schedulers is upper bounded by $|SA|^t$. In each iteration, we eliminate a class of schedulers that shares the same decision on the counterexample path. Therefore, the number of iterations $D \leq |SA|^t < \infty$. That is, the synthesis terminates in a finite steps and from the termination condition, we can guarantee that $\mathcal{M}||\mathcal{K} \models \phi$. For nonblocking, from our learning process, σ_{min} will always be preserved in the supervisor. Furthermore, from Assumption 1, we know σ_{min} will be defined for all possible paths. \square

3) *Complexity:* We define the size of an MDP $\mathcal{M}, S_{\mathcal{M}}$ as the total number of nondeterministic choices. In each

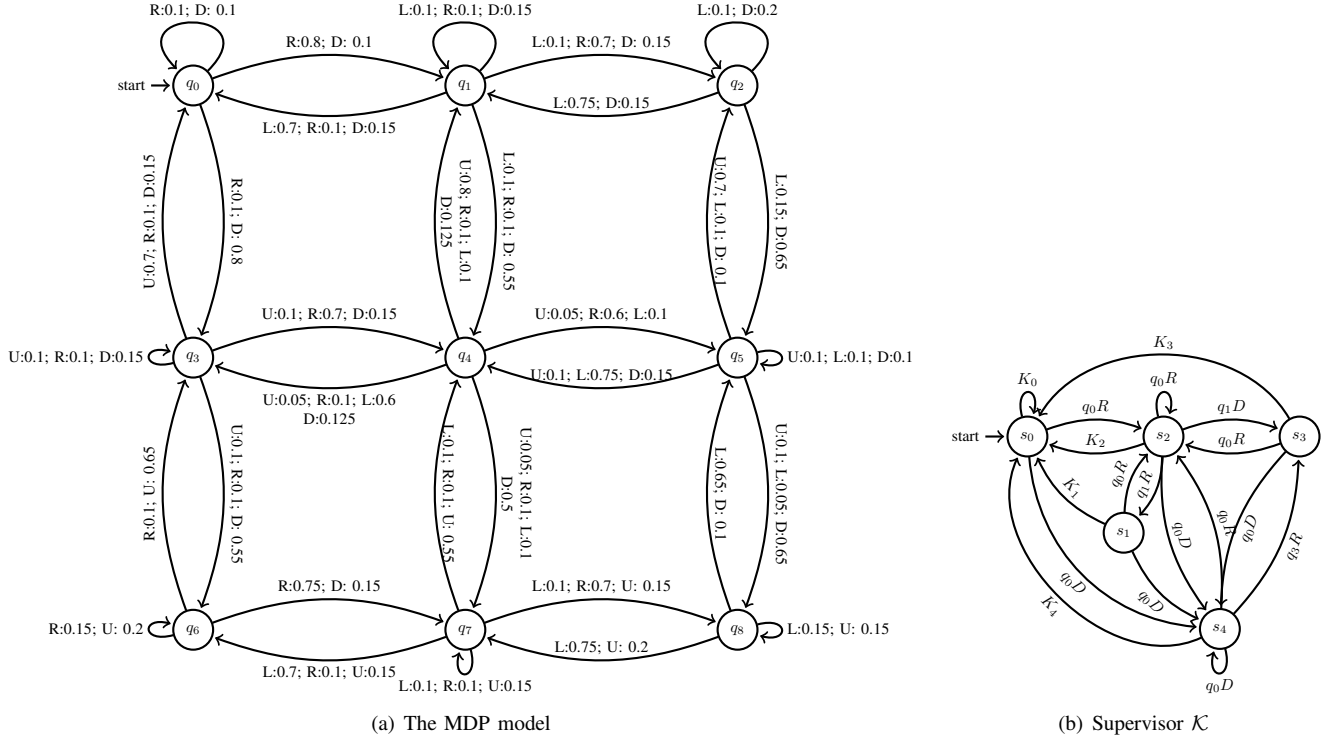


Fig. 1. The MDP model to synthesis and the resulting supervisor

iteration i , model checking has complexity polynomial with the size of the composed MDP \bar{M}^i which is upper bounded by $|SA|^t$ since we only consider all the possible actions in t steps. The counterexample selection algorithm has complexity exponential with t . It is also linear with the length of the PCTL formula L_f where L_f is defined to be the number of logical and temporal operators in the formula. The complexity for checking positive counterexample is bounded by $|SA|^t$. The number of iterations in the worst case is exponential with the time horizon t since there are at most $|SA|^t$ number of possible policies. As to L^* learning, the number of the states in the resulting DFA is upper bounded by $|SA|^t$ and the longest counterexample is bounded by t . Therefore to sum up, in the worst case our algorithm has a complexity polynomial to $|S|$ and $|A|$, exponential with t and linear with L_f .

4) *Illustrative example:* We give an illustrative example to show our learning based permissive supervisor synthesis framework. The system is as shown in Fig. 1 and we ignore the labeling for simplicity. It is essentially a robot with initial position at s_0 moving in a grid space. The action set is $\{U, D, L, R\}$ representing moving up, down, left and right. The number after the colon is the transition probability. For s_4 , we didn't draw its self loop transition due to the space limitation. But its transition probabilities can be easily inferred.

The probabilistic specification is $\phi = P_{\leq 0.6}(\varphi)$ where $\varphi = \text{true} U^{\leq 5} q_5$. To find the supervisor, we developed our own software toolkit which first does the model checking and gets the induced DTMC based on counterexample strategy. Then both DTMC and the specification (on this DTMC) is sent to COMICS to find the counterexample path which is then fed into L^* learning using libalf. Once the conjectured supervisor

is found, we parallel it with the original plant to see if the specification is satisfied. As in Algorithm 1, it runs iteratively until no counterexample is found.

For this particular example, the synthesis process runs for 4 iterations and the maximum probability to satisfy φ at the termination is 0.573053. The resulting supervisor has 5 states, as shown in Fig 1 where

- $K_0 = \{q_i a \mid \forall i > 0, (q_i, a, *) \in T\}$
- $K_1 = \{q_i a \mid \forall i > 0, (q_i, a, *) \in T\} \setminus \{q_2 D\}$
- $K_2 = \{q_i a \mid \forall i > 1, (q_i, a, *) \in T\} \cup \{q_1 L\}$
- $K_3 = \{q_i a \mid \forall i > 0, (q_i, a, *) \in T\} \setminus \{q_4 R\}$
- $K_4 = \{q_i a \mid \forall i > 0, (q_i, a, *) \in T\} \setminus \{q_3 R\}$

We also tested another case setting the probability threshold to 0.5 in the specification, this time the synthesis process run for 14 iterations, the maximum probability at the termination is 0.498412 and the resulting supervisor has 18 states. It is expected since the lower probability threshold can result in more counterexample strategies and thus more iterations are needed to eliminate them.

C. Multi-Agent Case

For simplicity, we assume that $N = 2$ but our framework can be readily extended to the case where $N > 2$.

1) *Learning based supervisor synthesis:* Algorithm 2 describes the flow of our framework. The whole work flow is similar to the single agent case, but with several key differences as described below.

The first difference is the model checking. Given an MAS consisting of N system models $\mathcal{M}_i = (S_i, s_0^i, A_i, T_i, L_i)$, $i = 1, \dots, N$, the initial local supervisors \mathcal{K}_i are trivial ones which allow every action. Then to avoid state space explosion, we

Algorithm 2: N-SPVSYN(\mathcal{M}, ϕ)

input : $\mathcal{M}_i = (S_i, s_{0,i}, A_i, T_i, L_i)$, $i = 1, \dots, N$,
probabilistic specification $\phi = \mathcal{P}_{\leq p}(\varphi)$
output: Local supervisors $\mathcal{K}_i, i \in [1, N]$ such that
 $\widetilde{\mathcal{M}} \models \phi$, where $\widetilde{\mathcal{M}} = \widetilde{\mathcal{M}}_1 || \widetilde{\mathcal{M}}_2 || \dots || \widetilde{\mathcal{M}}_N$ and
 $\widetilde{\mathcal{M}}_i = \mathcal{M}_i ||_{sup} \mathcal{K}_i$, $i = 1, \dots, N$.

```

1  $j \leftarrow 0, \mathcal{K}_i^0 = \text{DFA } \mathcal{G}_i$  such that  $\mathcal{L}(\mathcal{G}_i) = \Sigma_i^*, \widetilde{\mathcal{M}}_i^0 \leftarrow \mathcal{M}_i$ ;
2 while true do
3    $k \leftarrow -1$ ;
4   if ModelChecking( $\widetilde{\mathcal{M}}^j, \phi$ ) = false then
      $\pi^j \leftarrow$  counterexample path ;
      $k \leftarrow \text{SELECT}(\mathcal{M}, \pi^j)$  from [17];
      $\pi_k^j \leftarrow$  projection from  $\pi^j$ ;
     for  $i \in [1, N]$  do
       if  $i \neq k$  then
          $\pi_i^j \leftarrow$  positive counterexample;
          $\mathcal{K}_i^{j+1} \leftarrow \text{L* Learning}(\pi_i^j)$ ;
          $\widetilde{\mathcal{M}}_i^{j+1} \leftarrow \mathcal{M}_i || \mathcal{K}_i^{j+1}$ ;
       end
     if Positive or negative counterexamples exist then
        $j \leftarrow j + 1$ ;
     else
       Break;
     end
7 return  $\mathcal{K}_i^j, i \in [1, N]$ ;
```

will apply compositional model checking techniques to be introduced in Section IV-C2 to see if the given specification ϕ can be satisfied by the uncontrolled system. If the answer is yes and there is no positive counterexample, then we are done. Otherwise, counterexamples will be returned.

The second difference is the subsystem selection. After getting the returned counterexamples, it is time to determine which subsystem is at fault to such violation. The detail will be discussed in Section IV-C3. After identifying which subsystem \mathcal{M}_k actively causes the violation of the specification, its local supervisor will be refined. Other subsystem's supervisor will be refined based on the positive counterexamples. The refining process is based on L* learning algorithm with the returned counterexamples.

2) *Compositional model checking:* With multiple agents in the system, the total state space grows exponentially with N which makes the model checking computation prohibitively expensive. Therefore the conventional model checking methods will simply fail due to the high computation complexity. To prevent this, we propose to refer to compositional verification of the probabilistic systems in which the composed system is never built. The most well-known compositional technique is assume-guarantee reasoning (AGR). It has already been successfully applied to non-probabilistic model checking [31] and recently has been extended to probabilistic systems [15], [26], [20]. Here we use the asymmetric rules (ASYM) as shown below.

$$\frac{1 : \mathcal{M}_1 || A_b \models \phi \quad 2 : \mathcal{M}_2 \preceq A_b}{\mathcal{M}_1 || \mathcal{M}_2 \models \phi} \quad (1)$$

where A_b is called the assumption for \mathcal{M}_2 . Often the assumption is much smaller than the original system and the composition of \mathcal{M}_1 and A_b could be much smaller to alleviate the computation burden. The key problem for ASYM is to find a proper assumption and the main idea is to use counterexamples to refine the assumptions until the ASYM can answer the model checking problem.

While our framework has the flexibility in choosing different ASYM algorithms, we require the ASYM to be sound and complete with which a proper assumption can always be found for model checking purpose by showing the satisfaction of given specification or returning real counterexamples that witness the violation for the original system. For example, in [15], probabilistic automaton is used as the assumption form; with interpolation [32] based refinement algorithm, a sound and complete counterexample-guided abstraction refinement (CEGAR) framework is constructed with counterexample in the form of state action paths. With real counterexamples returned from the ASYM, the distributed supervisor can be revised accordingly.

3) *Counterexample and subsystem selection:* For probabilistic model checking of multiple plants, when the specification is not satisfied, the returned counterexamples belong to the composed system. After getting the counterexample paths, what's different from dealing with the single plant is that it is important to find out which subsystem *actively* performs the last action of each paths and causes the specification violation.

To do this, inspired by [17], we propose to partition the action set into *active*, *passive* and *normal* actions. Formally, for each subsystem i , we categorize the action set A_i into three disjoint sets $A_{i,a}$, $A_{i,p}$ and $A_{i,n}$, namely active, passive and normal actions and we only consider the *well-communicated systems* [17] as defined below.

Definition Given an MAS \mathcal{M} consisting of N system models $\mathcal{M}_i = (S_i, s_{0,i}^i, A_i, T_i, L_i)$, $i = 1, \dots, N$, we say \mathcal{M} is well communicated if the following hold:

- $\forall i \in [1, N]$, for each active action $a \in A_{i,a}$, there exists at least one corresponding passive action $a \in A_{j,p}, j \neq i$.
- $\forall i \in [1, N]$, for each passive action $a \in A_{i,p}$, there exists one and only one corresponding active action $a \in A_{j,a}, j \neq i$.

It is worth noting that such labeling only helps us to identify which subsystem is the cause of the violation. It is completely ignored in the model checking and supervisor synthesis stage. We adopt the SELECT algorithm (Algorithm 4) in [17]. Intuitively, we start from the end of the action sequence and find the subsystem that actively causes the last action to be performed. In this case, we can just focus on which subsystem \mathcal{M}_j performs the last active action of counterexample path ω . Once we find the subsystem \mathcal{M}_j , by projecting ω to the local action set of \mathcal{M}_j , we get a local counterexample path $\omega_{\mathcal{M}_j}$. Then it can be seen that the problem is to learn a local supervisor \mathcal{K}_j . For other subsystems that are not selected,

their supervisors remain unchanged. After that, we perform another round of model checking and refinement until no counterexample is generated.

4) *Computation Complexity, Correctness and Termination:*

In each iteration, for the framework we use in Section IV-C5, the model checking has the complexity upper bounded by $\prod_{i=1}^N |S_i A_i|^t$. It is also linear with the length of the PCTL formula L_f . The number of iterations is finite and in the worst case is exponential with the time horizon t and N since there are at most $\prod_{i=1}^N |S_i A_i|^t$ number of possible policies. As to L^* learning, the number of the states in the resulting DFA is upper bounded by $|SA|^t$ and the longest counterexample is bounded by t . Therefore in the worst case, our algorithm has a complexity exponential with t, N , polynomial with $S_{\mathcal{M}_i}, |S_i|, |A_i|$ and linear with L_f . However it is seen that in practice the compositional model checking rarely assume huge computation [20]. So this complexity analysis is rather conservative.

We then prove that the result of supervisor synthesis is correct by design and our framework always terminates.

Theorem 2. *The synthesized supervisors \mathcal{K}_i parallel composes with original MDP \mathcal{M}_i and the composed controlled system satisfies the specification ϕ . The whole synthesis process also terminates in finite iterations.*

Proof. From the soundness and completeness of ASYM rule with CEGAR, we know that our compositional model checking will always terminate and produce the correct answer. If $\mathcal{M} \models \phi$, the synthesis terminates when $i = 0$ and our theorem trivially holds. Otherwise in each iteration, counterexample selection algorithm will always terminate since we have only a finite number of agents. In the local supervisor synthesis, our modified L^* learning algorithm will always terminate in a finite number of quires [14]. So each iteration will terminate in finite time. Furthermore, the number of iterations is also finite since we are dealing with time bounded time properties. To sum up, the synthesis terminates in finite steps and from the termination condition and the soundness of AGAR, we can guarantee that $\mathcal{M} \models \phi$, where $\mathcal{M} = \mathcal{M}_1 || \mathcal{M}_2 || \dots || \mathcal{M}_N$ and $\mathcal{M}_i = \mathcal{M}_i || \mathcal{K}_i$, $i = 1, \dots, N$. \square

One of the conservativeness of the proposed method is that since we are using the compositional model checking, we are not able to get the minimum probability and the corresponding σ_{min} like in the single agent setting. It could be the case that at least one of the supervisors becomes blocking. Such scenario may be avoided in the run time by not executing the actions that lead to the blocking state. It also could be the case that the minimum probability to satisfy the property φ is larger than p . Then the model checking will keep on reporting counterexamples until the supervisors in one or more subsystems become blocking.

5) *An illustrative example:* Here we give an example to show the integration of our framework with assume-guarantee reasoning for MDPs. While our framework does not limit the choice of assume-guarantee reasoning algorithm to reduce the state space of MDPs, the counterexample guided abstraction refinement method developed in [15] is used for illustration

purpose in this example. The system consists of two MDP models. The first one is as shown in Fig. 2.

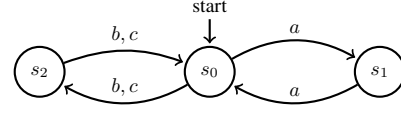


Fig. 2. \mathcal{M}_1 model

The system model \mathcal{M}_2 is extended from the example discussed in [15]. For \mathcal{M}_1 , the state set is $S_1 = \{0, 1, \dots, N-1\} \times \{bad, !bad\}$, $N > 2$ with initial state $\langle 0, !bad \rangle$; the action set is $A = \{a, b, c\}$. The transition function is given as follows. For $i < N-1$,

$$\begin{aligned} T(\langle i, !bad \rangle, a, \langle i+1, !bad \rangle) &= 0.9, \\ T(\langle i, !bad \rangle, a, \langle i, !bad \rangle) &= 0.1. \end{aligned}$$

For $i = N-1$

$$\begin{aligned} T(\langle N-1, !bad \rangle, a, \langle N-1, !bad \rangle) &= 0.9, \\ T(\langle N-1, !bad \rangle, a, \langle N-1, bad \rangle) &= 0.1. \end{aligned}$$

For action b , we have it define on the initial state with

$$\begin{aligned} T(\langle 0, !bad \rangle, b, \langle N-1, bad \rangle) &= 0.5, \\ T(\langle 0, !bad \rangle, b, \langle 0, !bad \rangle) &= 0.5. \end{aligned}$$

For \mathcal{M}_1 , $\{a, b\} \in A_{1,p}$ and $c \in A_{1,a}$. For \mathcal{M}_2 , $\{a, b\} \in A_{2,a}$ and $c \in A_{2,p}$. It is not hard to verify that $\{\mathcal{M}_1, \mathcal{M}_2\}$ compose a well-communicated system. $L_2(\langle N-1, bad \rangle) = \{failure\}$ and the specification is given as $\mathcal{P}_{\leq 0.3}[true \ U^{\leq N-1} failure]$.

Given the specification, our initial supervisors for both \mathcal{M}_1 and \mathcal{M}_2 enables all actions for any states. Then we have the initial abstract quotient automaton for \mathcal{M}_2 shown in Fig. 3. This quotient automaton is a probabilistic automaton [33] and can be seen as a special type of MDP where there may be many probability distributions defined for one action on a state. But the model checking problem on this quotient automaton is equivalent to MDP model checking [15]. In the first iteration, the model checking returns the counterexample path $(s_0, s_0^\#)c(s_2, s_2^\#)$. Since c is the active action in \mathcal{M}_1 . We select the first system to eliminate the projected path s_0cs_2 . In the second iteration, the model checking on the abstract

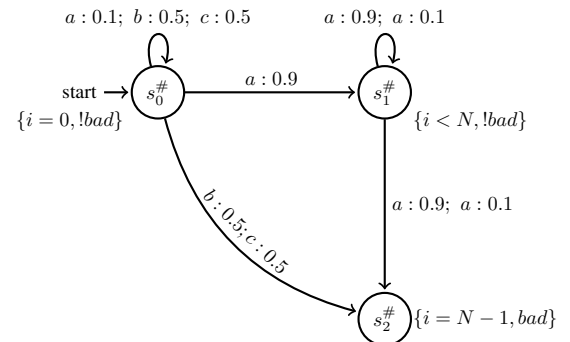


Fig. 3. The initial abstract quotient automaton $\mathcal{M}_2^\#$

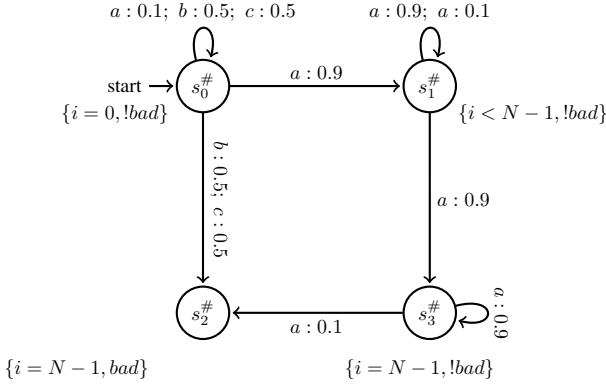


Fig. 4. The second abstract quotient automaton $\mathcal{M}_2^\#$.

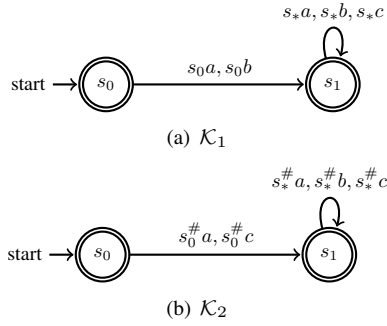


Fig. 5. The resulting supervisors \mathcal{K}_1 and \mathcal{K}_2 .

system first returns counterexample path as $s_0^\# a s_1^\# a s_2^\#$ with probability 0.81. Following algorithm in [15], we know this counterexample is introduced by the coarse abstraction and the refinement algorithm delivers new abstract system shown in Fig. 4. Then the model checking on the new abstract system returns counterexample path $(s_0 s_0^\#) b (s_2, s_2^\#)$ and this counterexample is a real counterexample. This time we refine the supervisor for \mathcal{M}_2 since b is its active action. After the second iteration, no more counterexample is generated and our resulting supervisors \mathcal{K}_1 and \mathcal{K}_2 are as shown in Fig. 5 where s_* and $s_*^\#$ represents any s and $s^\#$.

V. CONCLUSION

In this paper, we proposed an automated framework of learning based permissive supervisor synthesis for multi-agent systems. Assume-guarantee reasoning based model checking technique was applied to avoid the state space explosion problem. We also proposed to partition the action sets such that we know which subsystem is at fault when the specification is violated. It is guaranteed that we can get the correct supervisors in finite steps. For future research, we are interested in applying the same framework in partially known MDP models and more general class of specifications.

REFERENCES

- [1] J. J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker, *Mathematical techniques for analyzing concurrent and probabilistic systems*. American Mathematical Soc., 2004.
- [2] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.

- [3] A. Kučera and O. Stražovský, "On the controller synthesis for finite-state markov decision processes," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2005, pp. 541–552.
- [4] C. Baier, M. Größer, M. Leucker, B. Bollig, and F. Ciesinski, "Controller synthesis for probabilistic systems," in *Exploring New Frontiers of Theoretical Informatics*. Springer, 2004, pp. 493–506.
- [5] M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta, "Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 3227–3232.
- [6] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal aspects of computing*, vol. 6, no. 5, pp. 512–535, 1994.
- [7] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The complexity of decentralized control of markov decision processes," *Mathematics of operations research*, vol. 27, no. 4, pp. 819–840, 2002.
- [8] F. A. Oliehoek, M. T. Spaan, and N. Vlassis, "Optimal and approximate q-value functions for decentralized pomdps," *Journal of Artificial Intelligence Research*, vol. 32, pp. 289–353, 2008.
- [9] J. Filar and K. Vrieze, *Competitive Markov decision processes*. Springer Science & Business Media, 2012.
- [10] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *Formal Methods for Eternal Networked Software Systems*. Springer, 2011, pp. 53–113.
- [11] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer Science & Business Media, 2008.
- [12] M. Ujma, D. Parker, M. Kwiatkowska, V. Forejt, and K. Drager, "Permissive controller synthesis for probabilistic systems," *Logical Methods in Computer Science*, vol. 11, 2015.
- [13] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- [14] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.
- [15] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic cegar," in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 162–175.
- [16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.
- [17] S.-W. Lin and P.-A. Hsiung, "Counterexample-guided assume-guarantee synthesis through learning," *Computers, IEEE Transactions on*, vol. 60, no. 5, pp. 734–750, 2011.
- [18] J. Dai and H. Lin, "Automatic synthesis of cooperative multi-agent systems," in *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*. IEEE, 2014, pp. 6173–6178.
- [19] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 2015, pp. 239–248.
- [20] A. Komuravelli, C. S. Pășăreanu, and E. M. Clarke, "Assume-guarantee abstraction refinement for probabilistic systems," in *Computer aided verification*. Springer, 2012, pp. 310–326.
- [21] F. He, X. Gao, M. Wang, B.-Y. Wang, and L. Zhang, "Learning weighted assumptions for compositional verification of markov decision processes," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 21, 2016.
- [22] B. Wu and H. Lin, "Counterexample-guided permissive supervisor synthesis for probabilistic systems through learning," in *American Control Conference (ACC), 2015*. IEEE, 2015, pp. 2894–2899.
- [23] —, "Counterexample-guided distributed permissive supervisor synthesis for probabilistic multi-agent systems through learning," in *American Control Conference (ACC), 2016*. IEEE, 2016, pp. 5519–5524.
- [24] N. Jansen, E. Ábrahám, M. Volk, R. Wimmer, J.-P. Katoen, and B. Becker, "The comics tool—computing minimal counterexamples for dtmcs," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2012, pp. 349–353.
- [25] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, "libalf: The automata learning framework," in *Computer Aided Verification*. Springer, 2010, pp. 360–364.
- [26] R. Chadha and M. Viswanathan, "A counterexample-guided abstraction-refinement framework for Markov decision processes," *ACM Transactions on Computational Logic (TOCL)*, vol. 12, no. 1, p. 1, 2010.
- [27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

- [28] E. Ábrahám, B. Becker, C. Dehnert, N. Jansen, J.-P. Katoen, and R. Wimmer, “Counterexample generation for discrete-time Markov models: An introductory survey,” in *Formal Methods for Executable Software Models*. Springer, 2014, pp. 65–121.
- [29] R. Wimmer, N. Jansen, E. Ábrahám, B. Becker, and J.-P. Katoen, “Minimal critical subsystems for discrete-time markov models,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 299–314.
- [30] T. Han, J. Katoen, and D. Berteun, “Counterexample generation in probabilistic model checking,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 2, pp. 241–257, 2009.
- [31] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, “You assume, we guarantee: Methodology and case studies,” in *Computer Aided Verification*. Springer, 1998, pp. 440–451.
- [32] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, “Abstractions from proofs,” in *ACM SIGPLAN Notices*, vol. 39, no. 1. ACM, 2004, pp. 232–244.
- [33] R. Segala and N. Lynch, “Probabilistic simulations for probabilistic processes,” *Nordic Journal of Computing*, vol. 2, no. 2, pp. 250–273, 1995.